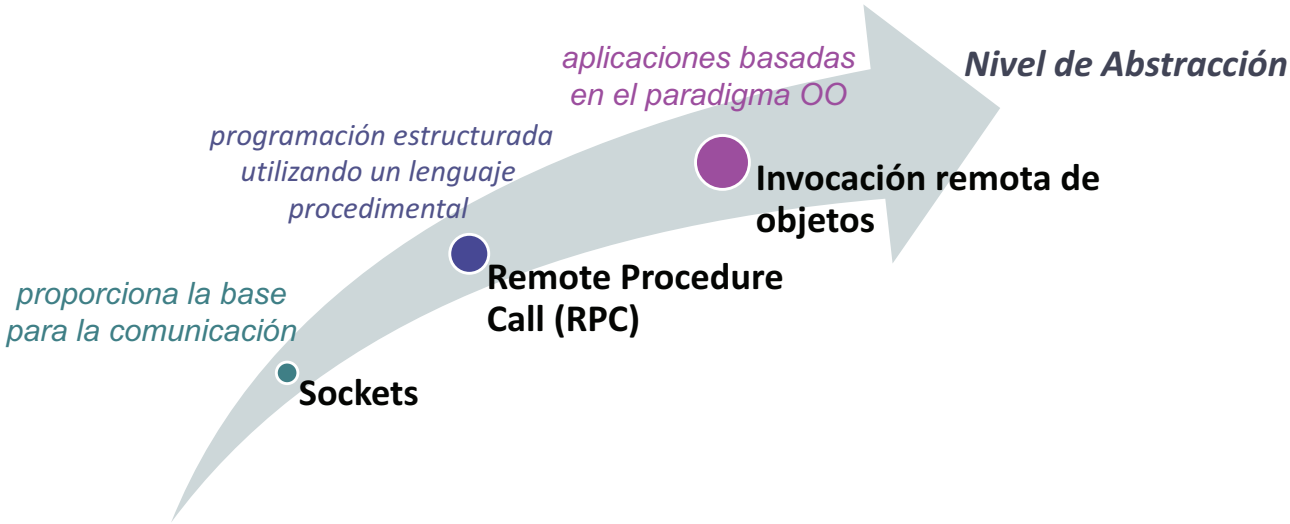
PROGRAMACION DISTRIBUIDA

Introducción a RMI (Remote Method Invocation)

Héctor Pérez

Abstracción de comunicaciones en sist. distribuidos



RMI (Remote Method Invocation) es la solución **Java** para la comunicación de objetos distribuidos

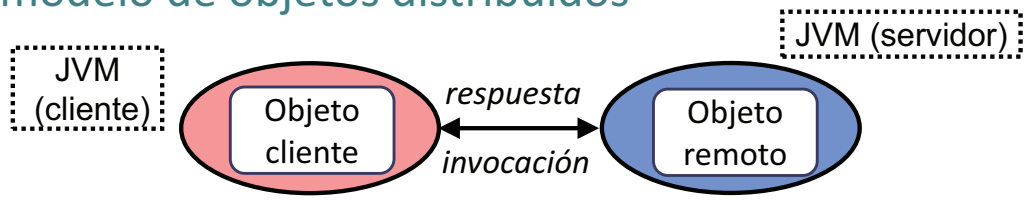
Necesidad de RMI

- Las clases *Socket* y *DatagramSocket* permiten implementar directamente aplicaciones distribuidas del tipo cliente/servidor
  - bajo nivel de abstracción (uso a nivel de bytes)
  - Clases nuevas y diferentes para clientes y servidores
  - alta complejidad en el mantenimiento
  - no está integrado dentro del paradigma OO de Java
- La respuesta de la comunidad Java fue *RMI*

*"RMI ha sido diseñado para hacer que la interacción entre objetos instanciados en dos particiones distribuidas de una aplicación Java que se ejecutan en diferentes JVM, se realice de forma **semejante** a como se hace entre objetos de una misma partición"*

RMI: Visión general

- Representa un mecanismo de invocación de objetos remotos como si fueran locales
  - modelo de objetos distribuidos



- Establece una **metodología**
  - extensión de clases, definición de interfaces ...
  - uso de herramientas adicionales (servicios de localización, gestores de seguridad, etc)

Objetivos de RMI

- Proporcionar un middleware para el desarrollo de aplicaciones distribuida manteniendo el estilo "Java":
  - Integra el **modelo de objetos distribuidos** en el lenguaje Java de una forma natural y manteniendo la semántica que le es propia.
  - Capacita para escribir aplicaciones distribuidas de una forma **simple**.
  - Mantiene y preserva en aplicaciones distribuidas el **tipado fuerte** propio de Java.
  - Proporciona diferentes modelos de **persistencia** de objetos distribuidos (objetos vivos, objetos persistentes, objetos con activación débil) para conseguir la escalabilidad de las aplicaciones.
  - Introduce los **niveles de seguridad** necesarios para garantizar la integridad de las aplicaciones distribuidas.

Ventajas e inconvenientes de RMI

- Ventaja:
  - Permite distribuir una aplicación de forma muy **transparente**, es decir, sin que el programador tenga que modificar apenas el código.
- Inconvenientes:
  - Sólo sirve para establecer aplicaciones distribuidas en las que todas las particiones de la aplicación están **codificadas en Java**.
  - Oculta los aspectos de la distribución, por lo que si una aplicación distribuida se diseña como si fuera a ser ejecutada en una única JVM puede hacerse muy **ineficiente**.
    - La interacción entre objetos remotos requiere internamente procesos de serialización de objetos, transmisión de mensajes, accesos a los gestores de seguridad, etc.

Aspectos que resuelve RMI (1/2)

- *¿Qué protocolos se utilizan para transferir la información al objeto remoto? ¿Y la información de retorno al cliente?*
- *¿Cómo se transfieren los argumentos y los valores de retorno?*
- *¿Cómo se conoce la ubicación de los objetos distribuidos?*
- *¿Cómo garantizar la integridad de las aplicaciones distribuidas?*

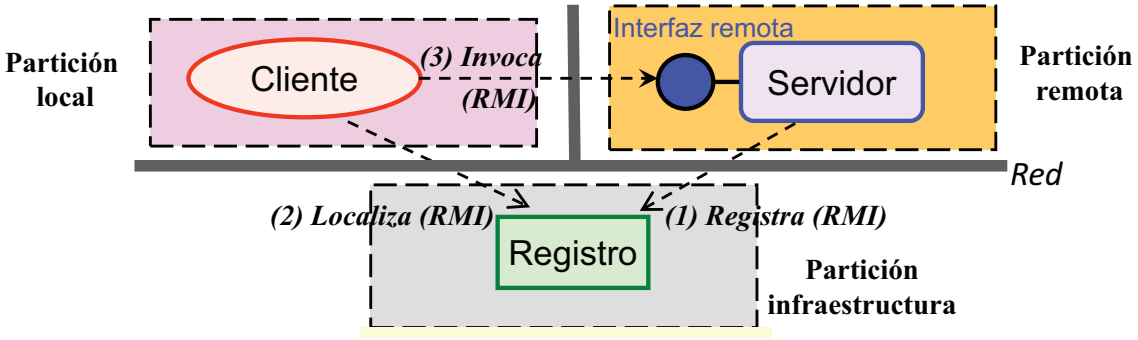
Aspectos que resuelve RMI (2/2)

- En el código para formular una aplicación distribuida se pueden identificas cinco aspectos diferentes:
  1. Código que implementa la funcionalidad de la aplicación. Es lo que llamamos **código de negocio**.
  2. Código de la **interfaz del cliente**. Es lo que hemos hecho con el *«clientProxy».*
  3. La **serialización** (marshalling) y reconstrucción (unmarshalling) de los datos que se intercambian. Es lo que hemos implementando utilizando *Streams*.
  4. Mecanismos para la invocación por **delegación** de los métodos del servidor. Corresponde a lo que hemos hecho con el código del *servant*.
  5. Módulos que **lanzan y configuran** las diferentes particiones de la aplicación. Es el código que hemos incluido en las clases principales de las particiones.
  6. Y **otro tipo de código** cuyo propósito es hace aplicaciones más robustas y escalables. Cosas como client-side caching, replicación de los servidores, servicios de localización de objetos, balanceo de carga, etc.

RMI automatiza su implementación

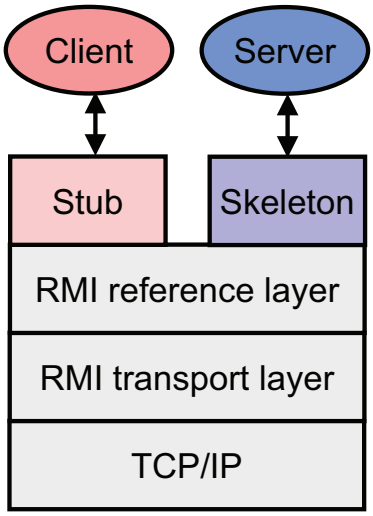
Componentes de aplicaciones distribuidas RMI

- **Clientes**: Conducen el flujo de la aplicación. ***Localizan*** e invocan métodos ofertados como remotos por los servidores.
- **Servidores**: Conjunto de objetos que ofrecen ***interfaces remotas públicas*** cuyos métodos pueden ser invocados clientes de cualquier procesador de la plataforma.
- **Registro**: Servicio estático que se establece en cada nudo, en el que se **registran** los servidores con un nombre, y donde los clientes los localizan.

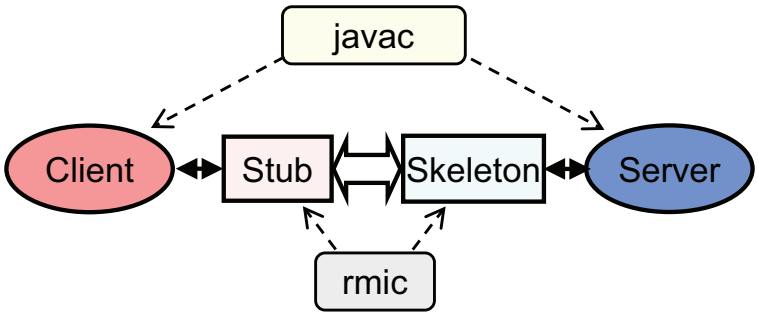


Arquitectura de RMI

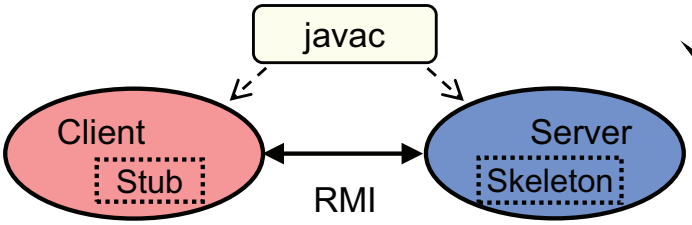
- Los ***stubs*** del cliente (proxy) y ***skeletons*** del servidor representan la interfaz entre la aplicación (código de negocio) y el resto del sistema RMI
  - proporcionan transparencia en la distribución
  - creados automáticamente
- El ***RMI reference layer*** se encarga de gestionar las invocaciones remotas (lado cliente y servidor)
- El ***RMI transport layer*** es responsable de gestionar todos los detalles de bajo nivel de la comunicación



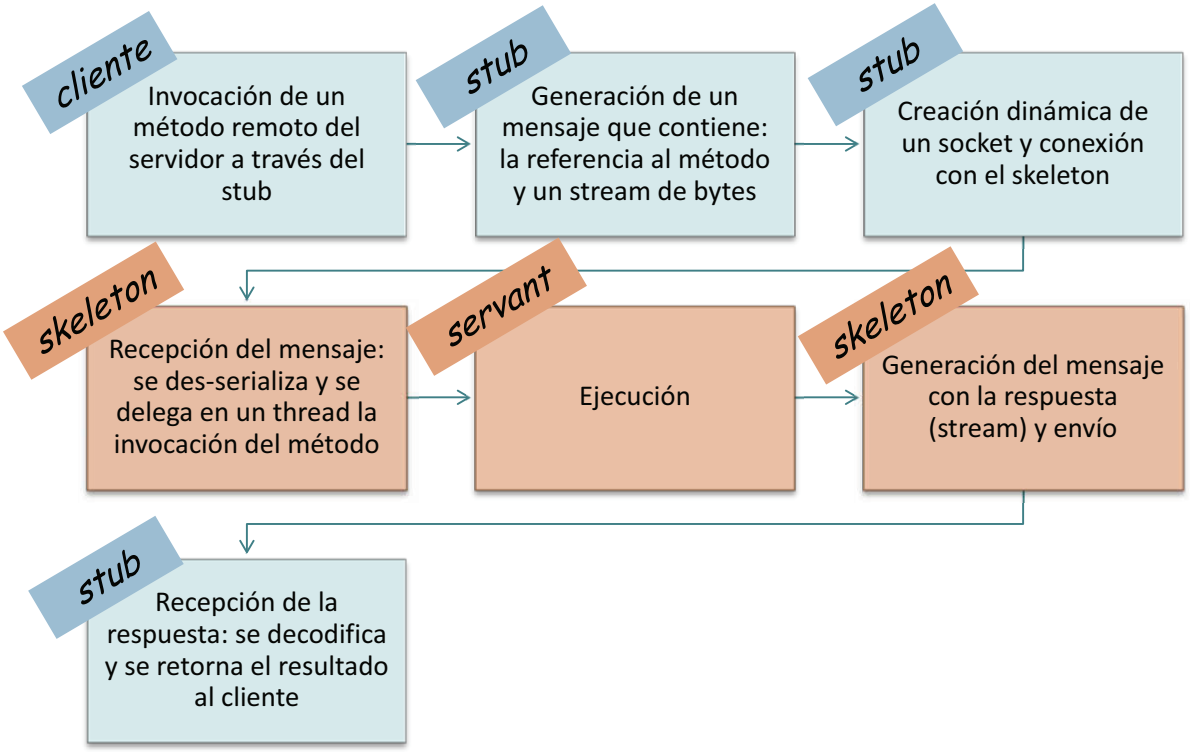
¡Advertencia!



Versión Java 1.5 o superior



Arquitectura de RMI: Stubs y skeletons



Invocaciones distribuidas y locales (1/2)

- **Semejanzas**:
  - Un objeto remoto puede ser pasado como ***parámetro*** de un método, y devuelto como resultado de los métodos.
  - El tipo de una referencia a un objeto remoto puede ser transformado por operaciones de ***casting,*** siempre que sean compatibles con sus relaciones de herencias.
  - A las referencias remotas se les puede aplicar el método ***instanceof***() para identificar dinámicamente las interfaces que soporta.

Invocaciones distribuidas y locales (2/2)

- **Diferencias**:
  - Los clientes interaccionan con los objetos remotos a través de las **interfaces remotas**, no a través de interfaces estándares.
  - Los objetos que se pasan como parámetros de métodos remotos se pasan **por referencia** y nunca por valor.
    - Por el contrario, los objetos que se pasan como parámetros de métodos que no son remotos se pasan **por copia** (valor) y nunca por referencia.
  - La semántica de los métodos heredados de *Object* es especializada para los objetos remotos, que son extensiones de ***RemoteObject***.
  - Las invocaciones de objetos remotos pueden lanzar **excepciones adicionales** que son propias de los mecanismos de comunicación.

Despliegue y localización de objetos remotos (1/2)

- Se busca proporcionar **independencia** entre el diseño de la aplicación y la formulación del despliegue
- RMI utiliza la estrategia de un servidor de nombres llamado ***registry*** como base del despliegue:
  - los servidores y clientes conocen **la dirección y el puerto del registro**
  - los servidores se registran en él a través de un identificador, el cuál queda asociado a una "referencia remota"
  - los clientes que quieren comunicarse con un servidor deben obtener su referencia remota a través del *registry.* Para ello, utilizan el identificador del servidor con el que quieren contactar

Despliegue y localización de objetos remotos (2/2)

- Ventajas en el uso del *registry* (servicio de nombres)
  - desacoplo de la ubicación del servidor
    - El registro se puede ubicar en una dirección fija y conocida del sistema distribuido, mientras que el servidor puede **cambiar su ubicación** sin afectar a los clientes
  - el registro es una aplicación sencilla, robusta y estable. Además, su mantenimiento es menor.
- No es la única estrategia de despliegue y localización
  - Por ejemplo, el estándar de distribución DDS utilizará la estrategia "*Anunciación/Descubrimiento*".

Proceso de invocación remota en RMI

El servidor debe registrarse (*bind*) en el *registry* bajo un nombre

El cliente localiza (*lookup*) la referencia de servidor en el *registry* por su nombre

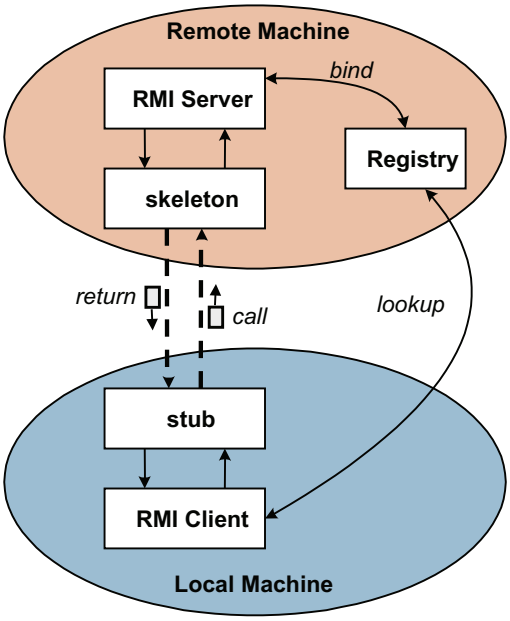
El cliente invoca localmente el *stub*

El *stub* transfiere como un mensaje la invocación al *skeleton*

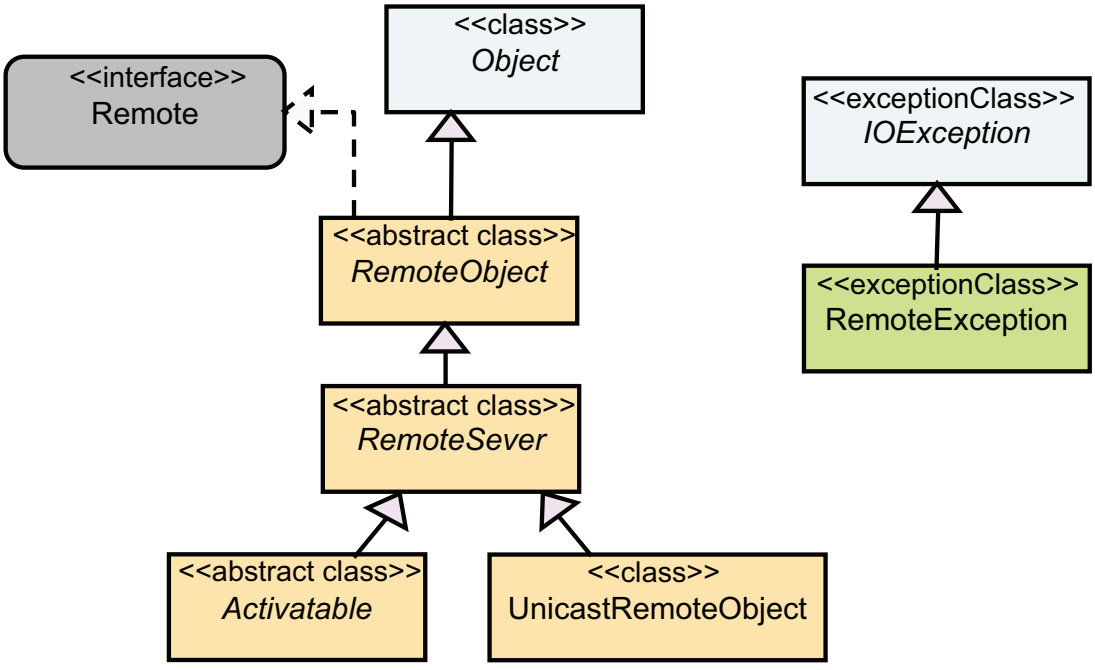
El *skeleton* invoca localmente el método del servidor

El *skeleton* transfiere al *stub* como mensaje los resultados obtenidos de la invocación

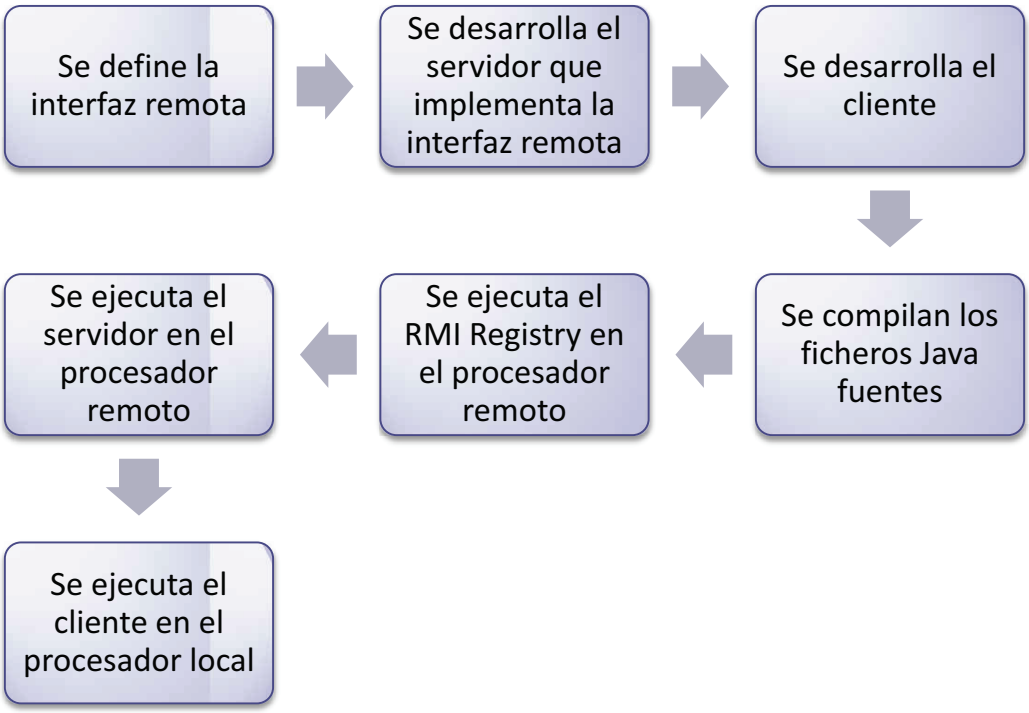
El *stub* finaliza la invocación del cliente retornándole los resultados



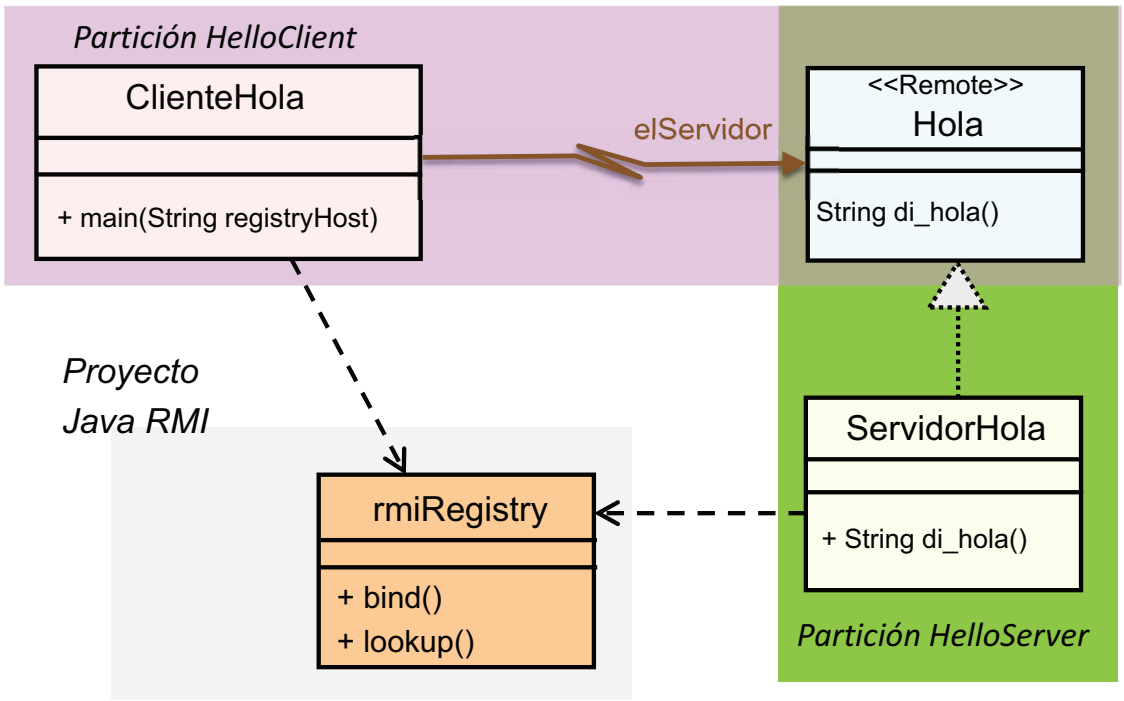
Interfaces y clases raíces definidas en RMI



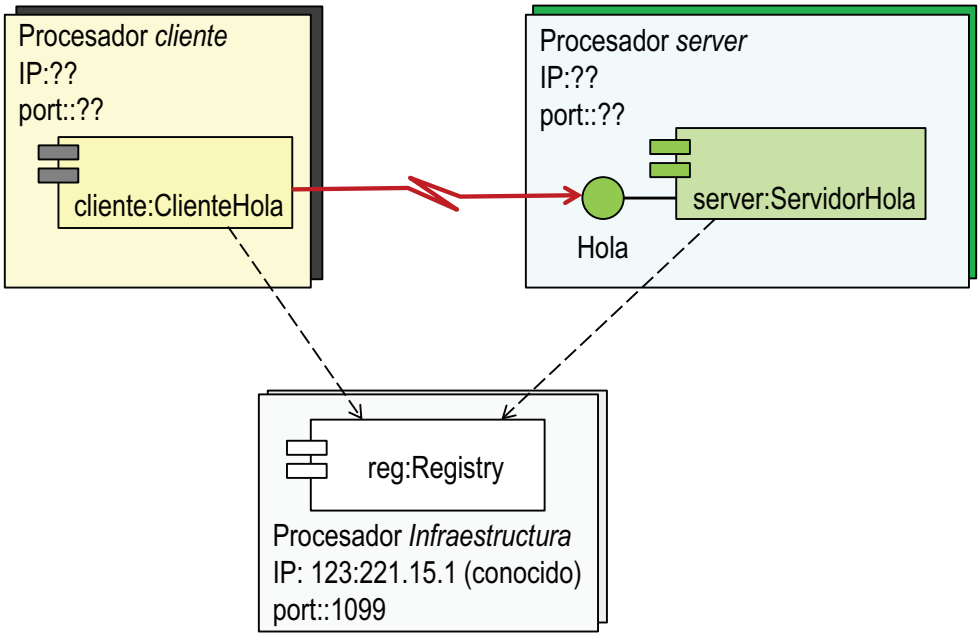
Pasos para desarrollar una aplicación distribuida RMI



Ejemplo HolaMundo (Diseño)



Ejemplo HolaMundo (Implementación)



HolaMundo (1/7): Interfaz remota (fichero *Hola.java*)

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hola extends Remote {
        String di_hola() throws RemoteException;
}
```

La interface que declara los métodos de un servidor que pueden ser invocados remotamente debe extender a la interface *Remote*

Todos los métodos que pueden ser invocados remotamente deben declarar la posibilidad de lanzar una *RemoteException*

HolaMundo (2/7): Servidor remoto (fichero *ServidorHola.java*)

```
import java.rmi.RemoteException;
import java.rmi.registry.*;
…

public class ServidorHola implements Hola {

    public ServidorHola() {}                                    // Constructor
    public String di_hola() { return "Hola, Mundo!"; }          // Implementación del método remoto

    public static void main(String args[]) {
      try {
        ServidorHola server = new ServidorHola();               //Instancia del servidor
        // El servidor se registra en el RMI como un objeto remoto y se obtiene su referencia remota
        Hola ServidorHolaRef = (Hola) …
        // El servidor se registra en el registry con un nombre (“Servidor_Hola”)
        Registry registry = …
        System.err.println("ServidorHola instalado");           // Servidor instalado con éxito
      } catch (Exception e) {
        System.err.println("Server exception: " + e.toString())
      }
    }
}
```


HolaMundo (3/7): Cliente (fichero *ClienteHola.java)*

```
import java.rmi.registry.*;

public class ClienteHola {
    private ClienteHola() {}                    // Constructor
    public static void main(String[] args) {    // El primer parámetro es el host de Registry
      String elHost=null;                       // Si no hay parámetro, usa el local
      if (args.length >= 1) elHost= args[0];
      try {
            // Se localiza el servidor en el registro por su nombre “Servidor_Hola” ·
            Registry registry = …
            Hola elServidor = (Hola) …
            String respuesta = elServidor.di_hola();        // Se invoca el servicio remoto
            System.out.println("Respuesta:  " + respuesta);
       } catch (Exception e) {
            System.err.println("Excepción del cliente: " + e.toString());}
    }
}
```

HolaMundo (4/7): Compilación de los ficheros Java

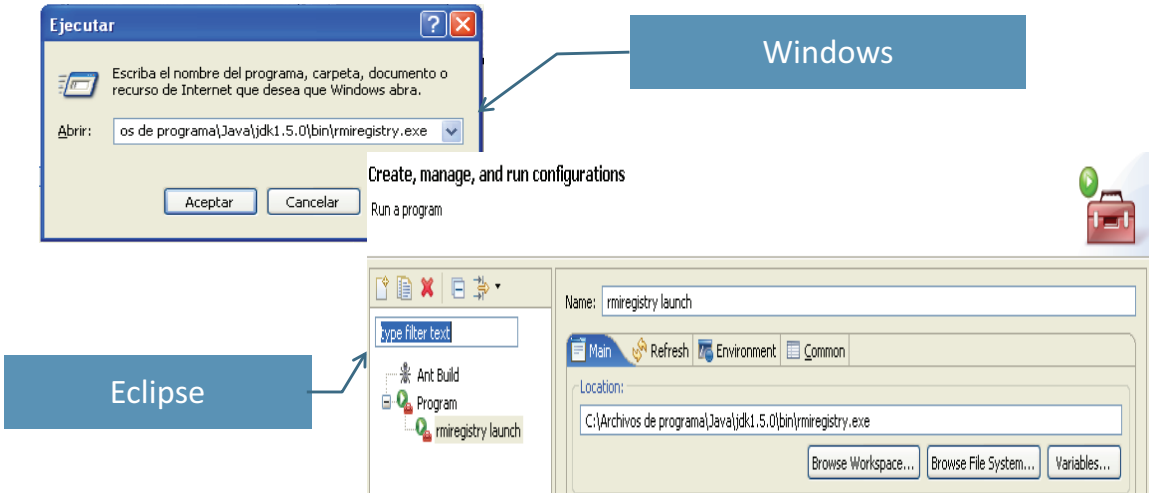
Se compila de la forma habitual

(con **javac**)

- *Ficheros del servidor*
  - ServidorHola.java
  - Hola.java
- *Ficheros del cliente*
  - ClienteHola.java
  - Hola.java

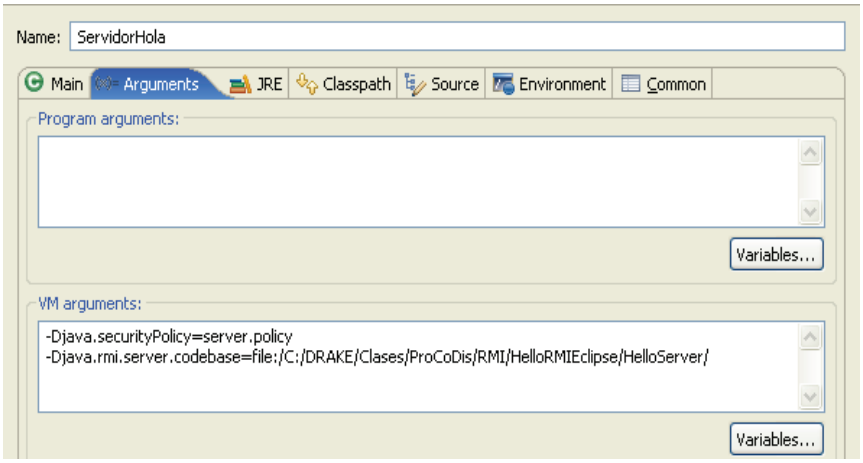
HolaMundo (5/7): Instanciación del RMI Registry

- Se lanza la ejecución del ***rmiregistry*** como un proceso en el procesador (por ejemplo, el del servidor)
  - Hay que ejecutar *independientemente* la aplicación **rmiregistry.exe** que existe en el jdk de java.



HolaMundo (6/7): Ejecución del servidor

- El servidor se ejecuta en el procesador remoto
  - Hay que establecer propiedades de JVM



  - Hay que establecer una política de seguridad adecuada

HolaMundo (7/7): Ejecución del Cliente

- El cliente se ejecuta en el procesador local